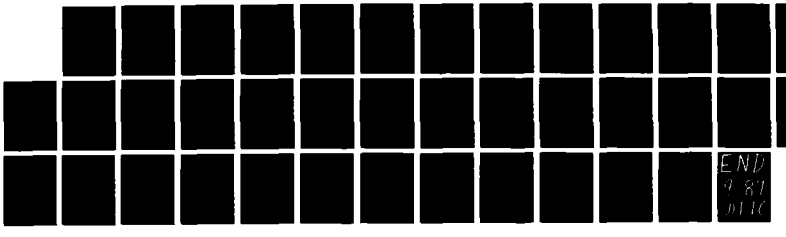
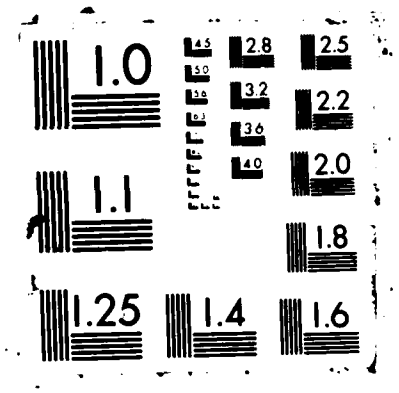
AD-A103 671 ADA (TRADEMARK) COMPILER VALIDATION SUMMARY REPORT 1/1
INFORMATIQUE INTERNATI..(U) ASSOCIATION FRANCAISE DE
NORMALISATION PARIS-LA-DEFENSE 06 MAY 87
UNCLASSIFIED F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

DTIC FILE COPY

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETEING FORM

1. REPORT NUMBER | 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Ada Compiler Validation Summary Report: Informatique Internationale SP-ADA, version 5.41 BULL SPS7/70	5. TYPE OF REPORT & PERIOD COVERED 6 May 1987 to 6 May 1988 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) AFNOR	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS AFNOR Tour Europe, Cedex 7 92080 Paris la Defense FRANCE	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081ASD/SIOL	12. REPORT DATE 6 May 1987 13. NUMBER OF PAGES 37
14. MONITORING AGENCY NAME & ADDRESS*(If different from Controlling Office)* AFNOR	15. SECURITY CLASS *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20. If different from Report)*

UNCLASSIFIED

DTIC ELECTE AUG 1 2 1987 E

18. SUPPLEMENTARY NOTES

19. KEYWORDS *(Continue on reverse side if necessary and identify by block number)*

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

See Attached.

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

AD-A183 671

AVF Control Number : AFNOR-VSR-8701

Ada* COMPILER
VALIDATION SUMMARY REPORT:

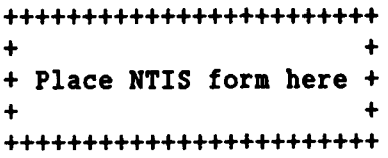
Informatique Internationale
SP-ADA, version 5.41
BULL SPS7/70

Completion of On-Site Validation:
6 May, 1987

Prepared by:
AFNOR
Tour Europe, cedex 7
92080 Paris la Défense
FRANCE

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C.

*Ada is a registered trademark of the United States Government (Ada Joint Program Office).


Place NTIS form here

Ada* Compiler Validation Summary Report:

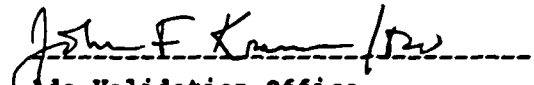
Compiler Name: SP-ADA, version 5.41

Host :	Target :
BULL SPS7/70 under SPIX version 22.0	BULL SPS7/70 under SPIX version 22.0

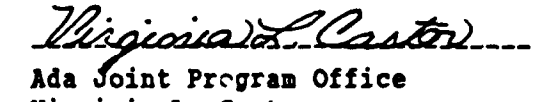
Testing Completed 6 May, 1987 Using ACVC 1.8

This report has been reviewed and is approved.

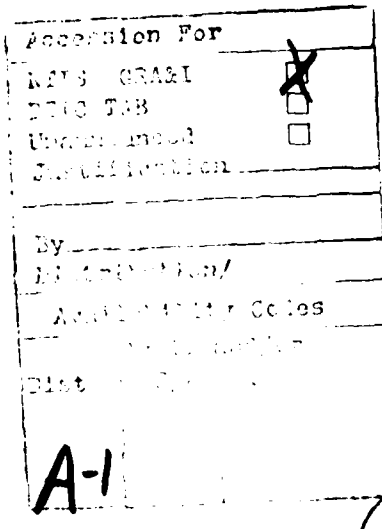
AFNOR
Dr. Jacqueline Sidi
Tour Europe, cedex 7
92080 Paris la Défense
FRANCE



Ada Validation Office
Dr.John F. Kramer
Institute for Defense Analyses
Alexandria VA



Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
USA - Washington DC



*Ada is a registered trademark of the United States Government (Ada Joint Program Office).

87

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the SP-ADA, version 5.41, using Version 1.8 of the Ada* Compiler Validation Capability (ACVC). The SP-ADA is hosted on a BULL SPS7/70 operating under SPIX, version 22.0. Programs processed by this compiler may be executed on a BULL SPS7/70 operating under SPIX, version 22.0.

On-site testing was completed on 6 May, 1987 at Informatique Internationale Sophia-Antipolis, under the direction of the AFNOR (AVF), according to Ada Validation Organization (AVO) policies and procedures. The AVF indentified 2210 of the 2399 tests in ACVC Version 1.8 to be processed during on-site testing of the compiler. The 19 tests withdrawn at the time of validation testing, as well as the 170 executable tests that make use of floating-point precision exceeding that supported by the implementation, were not processed. After the 2210 tests were processed, results for Class A, C, D, or E tests were examined for correct execution. Compilation listings for Class B tests were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors. There were 8 of the processed tests determined to be inapplicable ; The remaining 2202 tests were passed.

The results of validation are summarized in the following table:

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	102	252	334	244	161	97	138	261	130	32	218	233	2202
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	14	73	86	3	0	0	1	1	0	0	0	0	178
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

The AVF concludes that these results demonstrate acceptable conformity to ANSI/MIL-STD-1815A Ada.

*Ada is a registered trademark of the United States Government (Ada Joint Program Office).

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from characteristics of particular operating systems, hardware, or implementation strategies. All of the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any unsupported language constructs required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by the AVF according to policies and procedures established by the Ada Validation Organization (AVO). On-site testing was terminated on 6 May, 1987 at Valbonne.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
USA - Washington DC 20301-3081

or from:

AFNOR
Tour Europe, cedex 7
92080 - Paris la défense
FRANCE

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard
USA - Alexandria VA 22311

1.3 REFERENCES

1 Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, FEB 1983.

2. Ada Validation Organization: Policies and Procedures, MITRE Corporation, JUN 1982, PB 83-110601.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., DEC 1984.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. A set of programs that evaluates the conformity of a compiler to the Ada language specification, ANSI/MIL-STD-1815A.
Ada Standard	ANSI/MIL-STD-1815A, February 1983.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. In the context of this report, the AVF is responsible for conducting compiler validations according to established policies and procedures.
AVO	The Ada Validation Organization. In the context of this report, the AVO is responsible for setting procedures for compiler validations.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	A test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.

Inapplicable test	A test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	A test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn	A test found to be incorrect and not used to check conformity to the Ada language specification. A test may not be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. However, no checks are performed during execution to see if the test objective has been met. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of these units is checked by a set or executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attemped.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be

customized according to implementation-specific values --for example, an illegal file name. A list of the values used for this validation are listed in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: SP-ADA, version 5.41

ACVC Version: 1.8

Compiler option set: Direct code generation for floating-point arithmetic coprocessor (Motorola MC68881)

Certificate Expiration Date:

Host&Target Computer (the compiler is self-hosted):

Machine:	BULL SPS7/70
Operating System:	SPIX version 22.0
Memory Size:	8 MegaBytes
Disk space:	2 * 56 Megabytes

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. This compiler is characterized by the following interpretations of the Ada Standard:

. Capacities.

The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests) and D29002K.)

. Universal integer calculations.

An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAXINT. This implementation does not reject such calculations and processes them correctly. (See tests D4A002A, D4A002B, D4A004A, and D4A004B).

. Predefined types.

This implementation supports the additional predefined types TINY_INTEGER, SHORT_INTEGER, SHORT_FLOAT in the package STANDARD. (See tests B86001C and B86001D.)

. Based literals.

An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT.

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array type is declared (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array subtype is declared (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned.

Alternately, an implementation may accept the declaration. However, lengths must match array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Discriminated types.

  During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications during compilation. (See test E38104A.)

  In assigning record types with discriminants, the expression does not always appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype: it appears for one dimensional array types and for record types with discriminants; it does not appear for two dimensional array types. (See test C52013A.)

- Aggregates.

  In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

  In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43207B.)

  All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

. Functions

An implementation may allow the declaration of a parameterless function and an enumeration literal having the same profile in the same immediate scope, or it may reject the function declaration. If it accepts the function declarations, the use of the enumeration literal's identifier denotes the function. This implementation rejects the declarations. (See test E66001D.)

. Representation clauses.

The Ada Standard does not require an implementation to support representation clauses. If a representation clause is not supported, then the implementation must reject it. While the operation of representation clauses is not checked by Version 1.8 of the ACVC, they are used in testing other language features. This implementation accepts 'STORAGE_SIZE, 'SIZE and 'SMALL clauses; Enumeration representation clauses, including those that specify noncontiguous values, appear to be supported. (See tests C55B16A, C87B62A, C87B62B, C87B62C, and BC1002A.)

. Pragmas

The pragma INLINE is supported for procedures and for functions (See tests CA3004E and CA3004F.)

. Input/output.

The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants. The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, AE2101H, CE2201D, CE2201E, CE2401D.)

An existing text file can be opened in OUT_FILE mode, can be created in OUT_FILE mode, and can be created in IN_FILE mode. (See tests EE3102C.)

More than one internal file can be associated with each external file for text I/O for both reading and writing. (See tests CE3111A..E (5 tests).)

More than one internal file can be associated with each external file for sequential I/O for both reading and writing. (See test CE2107A..F (6 tests).)

More than one internal file can be associated with each external file for direct I/O for both reading and writing. (See tests CE2107A..F (6 tests).)

An external file associated with more than one internal file can be deleted. (See test CE2110B.)

Temporary sequential files are given a name. Temporary direct files are given a name. Temporary files given names are deleted when they are closed. (See tests CE2108A and CE2108C.)

- Generics.

Generic subprogram declarations and bodies (including subunits) can be compiled in separate compilations. (See test CA2009F.)

Generic package declarations and bodies (including subunits) can be compiled in separate compilations. (See test CA2009C and BC3205D.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.8 of the ACVC contains 2399 tests. When validation testing of SP-ADA was performed, 19 tests had been withdrawn. The remaining 2380 tests were potentially applicable to this validation. The AVF determined that 178 tests were inapplicable to this implementation, and that the 2202 applicable tests were passed by the implementation.

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	69	865	1192	17	13	46	2202
Failed	0	0	0	0	0	0	0
Inapplicable	0	2	176	0	0	0	178
Withdrawn	0	7	12	0	0	0	19
TOTAL	69	874	1380	17	13	46	2399

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	102	252	334	244	161	97	138	261	130	32	218	233	2202
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	14	73	86	3	0	0	1	1	0	0	0	0	178
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

3.4 WITHDRAWN TESTS

The following 19 tests were withdrawn from ACVC Version 1.8 at time of this validation:

C32114A	B33203C	C34018A
C35904A	B37401A	C41404A
B45116A	C48008A	B49006A
B4A010C	B74101B	C87B50A
C92005A	C940ACA	CA3005A..D (4 tests)
BC3204C		

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. For this validation attempt, 178 tests were inapplicable for the reasons indicated:

- C34001E, B52004D, B55B09C, and C55B07A use LONG_INTEGER which is not supported by this compiler.
- C34001G, and C35702B, use LONG_FLOAT which is not supported by this compiler.
- C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.

- C96005B checks implementations for which the smallest and largest values in type DURATION are different from the smallest and largest values in DURATION's base type. This is not the case for this implementation.

- The following 170 tests make use of floating-point precision that exceeds the maximum of 15 supported by the implementation:

C24113_K..Y (14 tests)
C35705_K..Y (14 tests)
C35706_K..Y (14 tests)
C35707_K..Y (14 tests)
C35708_K..Y (14 tests)
C35802_K..Y (14 tests)
C45241_K..Y (14 tests)
C45321_K..Y (14 tests)
C45421_K..Y (14 tests)
C45424_K..Y (14 tests)
C45521_K..Y (15 tests)
C45621_K..Y (15 tests)

3.6 SPLIT TESTS

If one or more errors do not appear to have been detected in a Class B test because of compiler error recovery, then the test is split into a set of smaller tests that contain the undetected errors. These splits are then compiled and examined. The splitting process continues until all errors are detected by the compiler or until there is exactly one error per split. Any Class A, Class C, or Class E test that cannot be compiled and executed because of its size is split into a set of smaller subtests that can be processed.

Splits were required for 19 Class B tests.

Some tests are split because the implementor choose not to continue semantic error processing for units that contain syntax errors; these tests contain syntax errors AND errors that are caught in semantics.

B24204A	B24204B	B24204C	B2A003A	B2A003B
B2A003C	B33301A	B37201A	B38008A	B41202A
B44001A	B64001A	B67001A	B67001B	B67001C
B67001D	B91003B	B95001A	B97102A	

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.8 produced by SP-ADA, was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of SP-ADA using ACVC Version 1.8 was conducted on-site by a validation team from the AVF. The configuration consisted of a BULL SPS7/70 operating under SPIX, version 22.0.

The prevalidation tests were used for validation; they were loaded by Informatique Internationale from a magnetic tape containing all tests provided by the AVF. Customization was done by Informatique Internationale. All tests were checked at prevalidation time.

After validation was performed the tests from the ACVC and from the validation run with compilation listings were checked for integrity by the AVF.

The full set of tests was compiled and linked on the BULL SPS7/70, and all executable tests were run on the BULL SPS7/70.

The compiler was tested using command scripts provided by Informatique Internationale and reviewed by the validation team.

Tests were compiled, linked, and executed (as approriate) using a single computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

The validation team departed from Informatique Internationale Sophia-Antipolis after testing was completed on 6 May, 1987.

The validation was performed on two machines : one for the chapters 2 to B, the other for chapters C, E and the non-applicable tests.

APPENDIX A

COMPLIANCE STATEMENT

Informatique Internationale has submitted the following compliance statement concerning the SP-ADA.

Compliance Statement

Configuration:

Compiler: SP-ADA, version 5.41

Test Suite: Ada* Compiler Validation Capability, Version 1.8

Host Computer:

Machine: BULL SPS7/70

Operating System: SPIX
version 22.0

Target Computer:

Machine: BULL SPS7/70

Operating System: SPIX
version 22.0

Informatique Internationale has made no deliberate extensions to the Ada language standard.

Informatique Internationale agrees to the public disclosure of this report.

Informatique Internationale agrees to comply with the Ada trademark policy, as defined by the Ada Joint Program Office.

Date: 3rd May 1987

Informatique Internationale

[illegible]
Centre de développement de Sophia-Antipolis
Les Cardoulines - Route des Dolines
06560 VALBONNE
Téléphone : 93.65.31.31 - Télex 970 332 F

---

*Ada is registered trademark of the United States Government (Ada Joint Program Office).

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of MIL-STD-1815A, and to certain allowed restrictions on representation classes. The implementation-dependent characteristics of the SP-ADA, version 5.41, are described in the following sections which discuss topics in Appendix F of the Ada Language Reference Manual (ANSI/MIL-STD-1815A).

Implementation-specific portions of the package STANDARD follow:

```
package STANDARD is

    ...

    type INTEGER       is range -2_147_483_648 .. 2_147_483_647;
    type SHORT_INTEGER is range        -32_768 .. 32_767;
    type TINY_INTEGER  is range           -128 .. 127;

    type  FLOAT        is digits 15 range -1.797_693_134_862_31E+308
                                       .. 1.797_693_134_862_31E+308;
    type SHORT_FLOAT   is digits 6  range -3.402_82E+38 .. 3.402_82E+38;

    type DURATION is delta 6.10351562500000E-5
         range -131072.00000 .. +131071.99993;
    ...

end STANDARD ;
```

1 IMPLEMENTATION-DEPENDENT PRAGMAS

1.1. SHARE_BODY Pragma

The SHARE_BODY pragma takes the name of a generic instantiation or a generic unit as the first argument and one of the identifiers TRUE or FALSE as the second argument. This pragma is only allowed immediately at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit.

When the first argument is a generic unit the pragma applies to all instantiations of that generic. When the first argument is the name of a generic instantiation the pragma applies only to the specified instantiation, or overloaded instantiations.

If the second argument is TRUE the compiler will try to share code generated for a generic instantiation with code generated for other instantiations of the same generic. When the second argument is FALSE each instantiation will get a unique copy of the generated code. The extent to which code is shared between instantiations depends on this pragma and the kind of generic formal parameters declared for the generic unit.

1.2.EXTERNAL_NAME Pragma

The EXTERNAL_NAME pragma takes the name of a subprogram or variable defined in Ada and allows the user to specify a different external name that may be used to reference the entity from other languages. The pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification.

1.3. INTERFACE_OBJECT Pragma

The INTERFACE_OBJECT pragma takes the name of a variable defined in another language and allows it to be referenced directly in Ada. The pragma will replace all occurrences of the variable name with an external reference to the second, link_argument. The pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification. The object must be declared as a scalar or an access type. The object cannot be any of the following:

a loop variable,
a constant,
an initialized variable,
an array, or
a record.

1.4. IMPLICIT_CODE Pragma

Takes one of the identifiers ON or OFF as the single argument. This pragma is only allowed within a machine code procedure. It specifies that implicit code generated by the compiler be allowed or disallowed. A warning is issued if OFF is used and any implicit code needs to be generated. The default is ON.

2. IMPLEMENTATION OF PREDEFINED PRAGMAS

2.1. CONTROLLED

This pragma is recognized by the implementation but has no effect.

2.2. ELABORATE

This pragma is implemented as described in Appendix B of the Ada Standard.

2.3. INLINE

This pragma is implemented as described in Appendix B of the Ada Standard.

2.4. INTERFACE

This pragma supports calls to 'C' and FORTRAN functions. The Ada subprograms can be either functions or procedures. The types of parameters and the result type for functions must be scalar, access or the predefined type ADDRESS in SYSTEM. An optional third argument overrides the default link name. All parameters must have mode IN. Record and array objects can be passed by reference using the ADDRESS attribute.

2.5. LIST

This pragma is implemented as described in Appendix B of the Ada Standard.

2.6. MEMORY_SIZE

This pragma is recognized by the implementation. The implementation does not allow SYSTEM to be modified by means of pragmas, the SYSTEM package must be recompiled.

2.7. OPTIMIZE

This pragma is recognized by the implementation but has no effect.

2.8. PACK

This pragma will cause the compiler to choose a non-aligned representation for composite types. Components that are smaller

than a STORAGE_UNIT are packed into a number of bits that is a power of two.

2.9. PAGE

This pragma is implemented as described in Appendix B of the Ada Standard.

2.10. PRIORITY

This pragma is implemented as described in Appendix B of the Ada Standard.

2.11. SHARED

This pragma is recognized by the implementation but has no effect.

2.12. STORAGE_UNIT

This pragma is recognized by the implementation. The implementation does not allow SYSTEM to be modified by means of pragmas, the SYSTEM package must be recompiled.

2.13. SUPPRESS

This pragma is implemented as described, except that RANGE_CHECK and DIVISION_CHECK cannot be suppressed.

2.14. SYSTEM_NAME

This pragma is recognized by the implementation. The implementation does not allow SYSTEM to be modified by means of pragmas, the SYSTEM package must be recompiled.

3. IMPLEMENTATION-DEPENDENT ATTRIBUTES

3.1. P'REF

For a prefix that denotes an object, a program unit, a label, or an entry:

This attribute denotes the effective address of the first of the storage units allocated to P. For a subprogram, package, task unit, or label, it refers to the address of the machine code associated with the corresponding body or statement. For an entry for which an address clause has been given, it refers to the corresponding hardware interrupt. The attribute is of the type

OPERAND defined in the package MACHINE_CODE. The attribute is only allowed within a machine code procedure.

(For a package, task unit, or entry, the 'REF attribute is not supported.)

4. SPECIFICATION OF PACKAGE SYSTEM

```
package SYSTEM
is
     type NAME is ( sps_unix );

     SYSTEM_NAME        : constant NAME := sps_unix;

     STORAGE_UNIT       : constant := 8;
     MEMORY_SIZE        : constant := 16_777_216;

     -- System-Dependent Named Numbers

     MIN_INT            : constant := -2_147_483_648;
     MAX_INT            : constant := 2_147_483_647;
     MAX_DIGITS         : constant := 15;
     MAX_MANTISSA       : constant := 31;
     FINE_DELTA         : constant := 2.0**(-30);
     TICK                       : constant := 0.01;

     -- Other System-dependent Declarations

     subtype PRIORITY is INTEGER range 0 .. 99;

     MAX_REC_SIZE : integer := 64*1024;

     type ADDRESS is private;

     NO_ADDR : constant ADDRESS;

     function PHYSICAL_ADDRESS(I: INTEGER) return ADDRESS;
     function ADDR_GT(A, B: ADDRESS) return BOOLEAN;
     function ADDR_LT(A, B: ADDRESS) return BOOLEAN;
     function ADDR_GE(A, B: ADDRESS) return BOOLEAN;
     function ADDR_LE(A, B: ADDRESS) return BOOLEAN;
     function ADDR_DIFF(A, B: ADDRESS) return INTEGER;
     function INCR_ADDR(A: ADDRESS; INCR: INTEGER) return ADDRESS;
     function DECR_ADDR(A: ADDRESS; DECR: INTEGER) return ADDRESS;

     function ">"(A, B: ADDRESS) return BOOLEAN renames ADDR_GT;
     function "<"(A, B: ADDRESS) return BOOLEAN renames ADDR_LT;
     function ">="(A, B: ADDRESS) return BOOLEAN renames ADDR_GE;
     function "<="(A, B: ADDRESS) return BOOLEAN renames ADDR_LE;
     function "-"(A, B: ADDRESS) return INTEGER renames ADDR_DIFF;
     function "+"(A: ADDRESS; INCR: INTEGER) return ADDRESS renames INCR_ADDR;
     function "-"(A: ADDRESS; DECR: INTEGER) return ADDRESS renames DECR_ADDR;
```

```
     pragma inline(PHYSICAL_ADDRESS);
     pragma inline(ADDR_GT);
     pragma inline(ADDR_LT);
     pragma inline(ADDR_GE);
     pragma inline(ADDR_LE);
     pragma inline(ADDR_DIFF);
     pragma inline(INCR_ADDR);
     pragma inline(DECR_ADDR);

private

     type ADDRESS is new integer;

     NO_ADDR : constant ADDRESS := 0;

end SYSTEM;
```

5. RESTRICTIONS ON REPRESENTATION CLAUSES

5.1. Pragma PACK

Array and record components that are smaller than a STORAGE_UNIT are packed into a number of bits that is a power of two. Objects and larger components are packed to the nearest whole STORAGE_UNIT.

5.2. Size Specification

The size specification T'SMALL is not supported except when the representation specification is the same as the value 'SMALL for the base type.

5.3. Record Representation Clauses

Components not aligned on even STORAGE_UNIT boundaries may not span more than four STORAGE_UNITs.

5.4. Address Clauses

Address clauses are supported for variables and constants.

5.5. Interrupts

Interrupt entries are supported for UNIX signals. The Ada for clause gives the UNIX signal number.

5.6. Representation Attributes

The ADDRESS attribute is not supported for the following entities:

Packages
Tasks
Labels
Entries

5.7. Machine Code Insertions

Machine code insertions are supported.

The general definition of the package MACHINE_CODE provides an assembly language interface for the target machine. It provides the necessary record type(s) needed in the code statement, an enumeration type of all the opcode mneumonics, a set of register definitions, and a set of addressing mode functions.

The general syntax of a machine code statement is as follows:

```
CODE_n'( opcode, operand {, operand} );
```

where n indicates the number of operands in the aggregate.

A special case arises for a variable number of operands. The operands are listed within a subaggregate. The format is as follows:

```
CODE_N'( opcode, (operand {, operand}) );
```

For those opcodes that require no operands, named notation must be used (cf. RM 4.3(4)).

```
CODE_0'( op => opcode );
```

The opcode must be an enumeration literal (i.e. it cannot be an object, attribute, or a rename).

An operand can only be an entity defined in MACHINE_CODE or the 'REF attribute.

The arguments to any of the functions defined in MACHINE_CODE must be static expressions, string literals, or the functions defined in MACHINE_CODE. The 'REF attribute may not be used as an argument in any of these functions.

Inline expansion of machine code procedures is supported.

6. INTERPRETATION OF EXPRESSIONS IN ADDRESS CLAUSES

Address clauses are supported for constants and variables. Interrupt entries are specified with the number of the UNIX signal.

7. RESTRICTIONS ON UNCHECKED CONVERSIONS

There is no restriction on Unchecked Conversions. However, a warning is given at compile time when sizes of source and target operands differ.

8. RESTRICTIONS ON UNCHECKED DEALLOCATIONS

None.

9. IMPLEMENTATION CHARACTERISTICS OF I/O PACKAGES

Instantiations of DIRECT_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITS) when the size of ELEMENT_TYPE exceeds that value. For example for unconstrained arrays such as string where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_REC_SIZE is defined in SYSTEM and can be changed by a program before instantiating DIRECT_IO to provide an upper limit on the record size. In any case the maximum size supported is 1024 x 1024 x STORAGE_UNIT bits. The instantiation of DIRECT_IO will raise USE_ERROR if MAX_REC_SIZE exceeds this absolute limit.

Instantiations of SEQUENTIAL_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITS) when the size of ELEMENT_TYPE exceeds that value. For example for unconstrained arrays such as string where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_RECORD_SIZE is defined in SYSTEM and can be changed by a program before instantiating SEQUENTIAL_IO to provide an upper limit on the record size. SEQUENTIAL_IO imposes no limit on MAX_REC_SIZE.

10. IMPLEMENTATION LIMITS

The following limits are actually enforced by the implementation. It is not intended to imply that resources up to or even near these limits are available to every program.

10.1. Line Length

The implementation supports a maximum line length of 500 characters including the end of line character.

10.2. Record and Array Sizes

The maximum size of a statically sized array type is 4_000_000 x STORAGE_UNITS. The maximum size of a statically sized record type is 4_000_000 x STORAGE_UNITS. A record type or array type declaration that exceeds these limits will generate a warning message.

10.3. Default Stack Size for Tasks

In the absence of an explicit STORAGE_SIZE length specification every task except the main program is allocated a fixed size stack of 10_240 STORAGE_UNITS. This is the value returned by T'STORAGE_SIZE for a task type T.

10.4. Default Collection Size

In the absence of an explicit STORAGE_SIZE length attribute the default collection size for an access type is 100_000 STORAGE_UNITS. This is the value returned by T'STORAGE_SIZE for an access type T.

10.5. Limit on Declared Objects

There is an absolute limit of 6_000_000 x STORAGE_UNITS for objects declared statically within a compilation unit. If this value is exceeded the compiler will terminate the compilation of the unit with a FATAL error message.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$BIG_ID1 Identifier the size of maximum input line length with varying last character.	RPT "(498, 'A')" & 1
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	RPT "(498, 'A')" & 2
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	RPT "(249, 'A')" & 3 & RPT "(249, 'A')"
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	RPT "(249, 'A')" & 4 & RPT "(249, 'A')"
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	RPT "(496, '0')" & 298

Name and Meaning	Value
$BIG_REAL_LIT A real literal that can be either of floating- or fixed-point type, has value 690.0, and has enough leading zeroes to be the size of the maximum line length.	RPT "(493, '0')" & 69.0E1
$BLANKS A sequence of blanks twenty characters fewer than the size of the maximum line length.	RPT "(479, ' ')"
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2_147_483_647
$EXTENDED_ASCII_CHARS A string literal containing all the ASCII characters with printable graphics that are not in the basic 55 Ada character set.	"abcdefghijklmnopqrstuvwxyz!$%?@[\]^`{}~"
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2_147_483_647
$FILE_NAME_WITH_BAD_CHARS An illegal external file name that either contains invalid characters, or is too long if no invalid characters exist.	"/illegal/file_name/2{]$%2102C.DAT"
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character, or is too long if no wild card character exists.	"/illegal/file_name/CE2102C*.DAT"
$GREATER_THAN_DURATION A universal real value that lies between DURATION'BASE'LAST and DURATION'LAST if any, otherwise any value in the range of DURATION.	100_000.0

TEST PARAMETERS

Name and Meaning	Value
$GREATER_THAN_DURATION_BASE_LAST The universal real value that is greater than DURATION'BASE'LAST if such a value exists.	10_000_000.0
$ILLEGAL_EXTERNAL_FILE_NAME1 An illegal external file name.	"/no/such/directory/ILLEGAL_EXTERNAL_FILE_NAME1"
$ILLEGAL_EXTERNAL_FILE_NAME2 An illegal external file name that is different from $ILLEGAL_EXTERNAL_FILE_NAME1.	"/no/such/directory/ILLEGAL_EXTERNAL_FILE_NAME2"
$INTEGER_FIRST The universal integer literal expression whose value is INTEGER'FIRST.	-2_147_483_648
$INTEGER_LAST The universal integer literal expression whose value is INTEGER'LAST.	2_147_483_647
$LESS_THAN_DURATION A universal real value that lies between DURATION'BASE'FIRST and DURATION'FIRST if any, otherwise any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST The universal real value that is less than DURATION'BASE'FIRST, if such a value exists.	-10_000_000.0
$MAX_DIGITS The universal integer literal whose value is the maximum digits supported for floating-point types.	15
$MAX_IN_LEN The universal integer literal whose value is the maximum input line length permitted by the implementation.	500

Name and Meaning	Value
$MAX_INT The universal integer literal whose value is SYSTEM.MAX_INT.	2_147_483_647
NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER if one exists, otherwise any undefined name.	TINY_INTEGER
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFD#
NON_ASCII_CHAR_TYPE An enumerated type definition for a character type whose literals are the identifier NON_NULL and all non-ASCII characters with printable graphics.	(NON_NULL)

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the ADA Standard. The following 19 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

. C32114A : An unterminated string literal occurs at line 62.

. B33203C : The reserved word "IS" is misspelled at line 45.

. C34018A : The call of function G at line 114 is ambiguous in the presence of implicit conversions.

. C35904A : The elaboration of subtype declarations SFX3 and SFX4 may raise NUMERIC_ERROR instead of CONSTRAINT_ERROR as expected in the test.

. B37401A : The object declarations at lines 126 through 135 follow subprogram bodies declared in the same declarative part.

. C41404A : The values of 'LAST and 'LENGTH are incorrect in if statements from line 74 to the end of the test.

. B45116A : ARRPRIBL1 and ARRPRIBL2 are initialized with a value of the wrong type--PRIBOOL_TYPE instead of ARRPRIBOOL_TYPE--at line 41.

. C48008A : The assumption that evaluation of default initial values occurs when an exception is raised by an allocator is incorrect according to AI-00397.

. B49006A : Object declarations at line 41 and 50 are terminated incorrectly with colons; and end case; is missing from line 42.

. B4A010C : The object declaration in line 18 follows a subprogram body of the same declarative part.

. B74101B : The begin at line 9 causes a declarative part to be treated as a sequence of statements.

. C87B50A : The call of "/=" at line 31 requires a use clause for package A.

. C92005A : The "/=" for type PACK.BIG_INT at line 40 is not visible without a use clause for package PACK.

. C940ACA : The assumption that allocated task TT1 will run prior to the main program, and thus assign SPYNUMB the value checked for by the main program, is erroneous.

. CA3005A..D (4 tests) : No valid elaboration order exists for these tests.

. BC3204C : The body of BC3204C0 is missing.

END

9-87

DTIC